FUNCTIONAL PEARL

Applicative programming with effects

CONOR MCBRIDE
University of Nottingham

ROSS PATERSON
City University, London

Abstract

In this paper, we introduce Applicative functors—an abstract characterisation of an applicative style of effectful programming, weaker than Monads and hence more widespread. Indeed, it is the ubiquity of this programming pattern that drew us to the abstraction. We retrace our steps in this paper, introducing the applicative pattern by diverse examples, then abstracting it to define the Applicative type class and introducing a bracket notation which interprets the normal application syntax in the idiom of an Applicative functor. Further, we develop the properties of applicative functors and the generic operations they support. We close by identifying the categorical structure of applicative functors and examining their relationship both with Monads and with Arrows.

1 Introduction

This is the story of a pattern that popped up time and again in our daily work, programming in Haskell (Peyton Jones, 2003), until the temptation to abstract it became irresistable. Let us illustrate with some examples.

Sequencing commands One often wants to execute a sequence of commands and collect the sequence of their responses, and indeed there is such a function in the Haskell Prelude (here specialised to IO):

```
sequence :: [IO a] → IO [a]
sequence []       = return []
sequence (c : cs) = do
  x  ← c
  xs ← sequence cs
  return (x : xs)
```

In the $(c : cs)$ case, we collect the values of some effectful computations, which we then use as the arguments to a pure function (:). We could avoid the need for names to wire these values through to their point of usage if we had a kind of 'effectful application'. Fortunately, exactly such a thing lives in the standard Monad library:

```
ap :: Monad m ⇒ m (a → b) → m a → m b
ap mf mx = do
  f ← mf
  x ← mx
  return (f x)
```

Using this function we could rewrite sequence as:

```
sequence :: [IO a] → IO [a]
sequence []       = return []
sequence (c : cs) = return (:) `ap` c `ap` sequence cs
```

where the return operation, which every Monad must provide, lifts pure values to the effectful world, whilst ap provides 'application' within it.

Except for the noise of the returns and aps, this definition is in a fairly standard applicative style, even though effects are present.

Transposing 'matrices' Suppose we represent matrices (somewhat approximately) by lists of lists. A common operation on matrices is transposition[1].

```
transpose :: [[a]] → [[a]]
transpose []       = repeat []
transpose (xs : xss) = zipWith (:) xs (transpose xss)
```

Now, the binary zipWith is one of a family of operations that 'vectorise' pure functions. As Daniel Fridlender and Mia Indrika (2000) point out, the entire family can be generated from repeat, which generates an infinite stream from its argument, and zapp, a kind of 'zippy' application.

```
repeat :: a → [a]
repeat x = x : repeat x
```

```
zapp :: [a → b] → [a] → [b]
zapp (f : fs) (x : xs) = f x : zapp fs xs
zapp _        _        = []
```

The general scheme is as follows:

$$\mathsf{zipWith}_n \;::\; (a_1 \to \cdots \to a_n \to b) \to [a_1] \to \cdots \to [a_n] \to [b]$$
$$\mathsf{zipWith}_n\ f\ xs_1\ \ldots\ xs_n = \mathsf{repeat}\ f\ `\mathsf{zapp}`\ xs_1\ `\mathsf{zapp}`\ \ldots\ `\mathsf{zapp}`\ xs_n$$

In particular, transposition becomes

```
transpose :: [[a]] → [[a]]
transpose []       = repeat []
transpose (xs : xss) = repeat (:) `zapp` xs `zapp` transpose xss
```

Except for the noise of the repeats and zapps, this definition is in a fairly standard applicative style, even though we are working with vectors.

Evaluating expressions When implementing an evaluator for a language of expressions, it is customary to pass around an environment, giving values to the free variables. Here is a very simple example

[1] This function differs from the one in the standard library in its treatment of ragged lists

```
data Exp v = Var v
           | Val Int
           | Add (Exp v) (Exp v)
eval :: Exp v → Env v → Int
eval (Var x)   γ = fetch x γ
eval (Val i)   γ = i
eval (Add p q) γ = eval p γ + eval q γ
```

where Env v is some notion of environment and fetch x projects the value for the variable x.

We can eliminate the clutter of the explicitly threaded environment with a little help from some very old friends, designed for this purpose:

```
eval :: Exp v → Env v → Int
eval (Var x)   = fetch x
eval (Val i)   = 𝕂 i
eval (Add p q) = 𝕂 (+) `𝕊` eval p `𝕊` eval q
```

where

$$\mathbb{K} :: a \to env \to a \qquad \mathbb{S} :: (env \to a \to b) \to (env \to a) \to (env \to b)$$
$$\mathbb{K}\ x\ \gamma = x \qquad \mathbb{S}\ ef\ es\ \gamma = (ef\ \gamma)\ (es\ \gamma)$$

Except for the noise of the $\mathbb{K}$ and $\mathbb{S}$ combinators[2], this definition of eval is in a fairly standard applicative style, even though we are abstracting an environment.

2 The Applicative class

We have seen three examples of this 'pure function applied to funny arguments' pattern in apparently quite diverse fields—let us now abstract out what they have in common. In each example, there is a type constructor f that embeds the usual notion of value, but supports its *own peculiar way* of giving meaning to the usual applicative language—its *idiom*. We correspondingly introduce the Applicative class:

```
infixl 4 ⊛
class Applicative f where
  pure :: a → f a
  (⊛) :: f (a → b) → f a → f b
```

This class generalises $\mathbb{S}$ and $\mathbb{K}$ from threading an environment to threading an effect in general.

We shall require the following laws for applicative functors:

identity	pure id ⊛ u	=	u
composition	pure $(\cdot)$ ⊛ u ⊛ v ⊛ w	=	u ⊛ (v ⊛ w)
homomorphism	pure f ⊛ pure x	=	pure (f x)
interchange	u ⊛ pure x	=	pure ($\lambda f \to f\ x$) ⊛ u

[2] also known as the return and ap of the environment Monad

The idea is that pure embeds pure computations into the pure fragment of an effectful world—the resulting computations may thus be shunted around freely, as long as the order of the genuinely effectful computations is preserved.

You can easily check that applicative functors are indeed functors, with the following action on functions:

$$(\langle\$\rangle) :: \mathsf{Applicative}\ f \Rightarrow (a \to b) \to f\ a \to f\ b$$
$$f \langle\$\rangle\ u = \mathsf{pure}\ f \circledast u$$

Moreover, any expression built from the Applicative combinators can be transformed to a canonical form in which a single pure function is 'applied' to the effectful parts in depth-first order:

$$\mathsf{pure}\ f \circledast u_1 \circledast \ldots \circledast u_n$$

This canonical form captures the essence of Applicative programming: computations have a fixed structure, given by the pure function, and a sequence of subcomputations, given by the effectful arguments. We therefore find it convenient, at least within this paper, to write this form using a special bracket notation,

$$[\![f\ u_1\ \ldots\ u_n]\!]$$

indicating a shift into the idiom of an Applicative functor, where a pure function is applied to a sequence of effectful arguments using the appropriate $\circledast$. Our intention is to give an indication that effects are present, whilst retaining readability of code.

Given Haskell extended with multi-parameter type classes, enthusiasts for overloading may replace '$[\![$' and '$]\!]$' by identifiers $\mathbb{i}$ and $\mathbb{I}$ with the right behaviour[3].

The IO monad, and indeed any Monad, can be made Applicative by taking pure = return and $(\circledast)$ = ap. We could alternatively use the variant of ap that performs the computations in the opposite order, but we shall keep to the left-to-right order in this paper. Sometimes we can implement the Applicative interface a little more directly, as with $(\to)\ env$:

```
instance Applicative ((→) env) where
  pure x  = λγ → x                -- 𝕂
  ef ⊛ ex = λγ → (ef γ) (ex γ)    -- 𝕊
```

With these instances, sequence and eval become:

```
sequence :: [IO a] → IO [a]
sequence []     = ⟦[]⟧
sequence (c : cs) = ⟦(:) c (sequence cs)⟧

eval :: Exp v → Env v → Int
eval (Var x)   = fetch x
eval (Val i)   = ⟦i⟧
eval (Add p q) = ⟦(+) (eval p) (eval q)⟧
```

If we want to do the same for our transpose example, we shall have to avoid the

[3] *Hint:* Define an overloaded function applicative $u\ v1\ \ldots\ vn\ \mathbb{i} = u \circledast v1 \circledast \ldots \circledast vn$

library's 'list of successes' (Wadler, 1985) monad and take instead an instance Applicative [] that supports 'vectorisation', where pure = repeat and (⊛) = zapp, yielding

```
transpose :: [[a]] → [[a]]
transpose      []       = ⟦[]⟧
transpose (xs : xss) = ⟦(:) xs (transpose xss)⟧
```

In fact, repeat and zapp are not the return and ap of any Monad.

3 Traversing data structures

Have you noticed that sequence and transpose now look rather alike? The details that distinguish the two programs are inferred by the compiler from their types. Both are instances of the *applicative distributor* for lists:

```
dist :: Applicative f ⇒ [f a] → f [a]
dist []       = ⟦[]⟧
dist (v : vs) = ⟦(:) v (dist vs)⟧
```

Distribution is often used together with 'map'. For example, given the monadic 'failure-propagation' applicative functor for Maybe, we can map some failure-prone operation (a function in $a \to$ Maybe b) across a list of inputs in such a way that any individual failure causes failure overall.

```
flakyMap :: (a → Maybe b) → [a] → Maybe [b]
flakyMap f ss = dist (fmap f ss)
```

As you can see, flakyMap traverses ss twice—once to apply f, and again to collect the results. More generally, it is preferable to define this applicative mapping operation directly, with a single traversal:

```
traverse :: Applicative f ⇒ (a → f b) → [a] → f [b]
traverse f    []       = ⟦[]⟧
traverse f (x : xs) = ⟦(:) (f x) (traverse f xs)⟧
```

This is just the way you would implement the ordinary fmap for lists, but with the right-hand sides wrapped in $[\![\cdots]\!]$, shifting them into the idiom. Just like fmap, traverse is a useful gadget to have for many data structures, hence we introduce the type class Traversable, capturing functorial data structures through which we can thread an applicative computation:

```
class Traversable t where
  traverse :: Applicative f ⇒ (a → f b) → t a       → f (t b)
  dist     :: Applicative f ⇒                 t (f a) → f (t a)
  dist     = traverse id
```

Of course, we can recover an ordinary 'map' operator by taking f to be the identity—the simple applicative functor in which all computations are pure:

```
newtype Id a = An{an :: a}
```

Haskell's **newtype** declarations allow us to shunt the syntax of types around without changing the run-time notion of value or incurring any run-time cost. The 'labelled field' notation defines the projection $\mathsf{an} :: \mathsf{Id}\ a \to a$ at the same time as the constructor $\mathsf{An} :: a \to \mathsf{Id}\ a$. The usual applicative functor has the usual application:

```
instance Applicative Id where
  pure          = An
  An f ⊛ An x = An (f x)
```

So, with the **newtype** signalling which Applicative functor to thread, we have

$$\mathsf{fmap}\ f = \mathsf{an} \cdot \mathsf{traverse}\ (\mathsf{An} \cdot f)$$

Meertens (1998) defined generic dist-like operators, families of functions of type $t\ (f\ a) \to f\ (t\ a)$ for every regular functor t (that is, 'ordinary' uniform datatype constructors with one parameter, constructed by recursive sums of products). His conditions on f are satisfied by applicative functors, so the regular type constructors can all be made instances of Traversable. The rule-of-thumb for traverse is 'like fmap but with $[\![\cdots]\!]$ on the right'. For example, here is the definition for trees:

```
data Tree a = Leaf | Node (Tree a) a (Tree a)

instance Traversable Tree where
  traverse f  Leaf          = ⟦Leaf⟧
  traverse f (Node l x r) = ⟦Node (traverse f l) (f x) (traverse f r)⟧
```

This construction even works for non-regular types. However, not every Functor is Traversable. For example, the functor $(\to)\ env$ cannot in general be Traversable. To see why, take $env = \mathsf{Integer}$ and try to distribute the Maybe functor!

Although Meertens did suggest that threading monads might always work, his primary motivation was to generalise reduction or 'crush' operators, such as flattening trees and summing lists. We shall turn to these in the next section.

4 Monoids are phantom Applicative functors

The data that one may sensibly accumulate have the Monoid structure:

```
class Monoid o where
  ∅   :: o
  (⊕) :: o → o → o
```

such that '$\oplus$' is an associative operation with identity $\emptyset$. The functional programming world is full of monoids—numeric types (with respect to zero and plus, or one and times), lists with respect to $[\,]$ and $+\!\!+$, and many others—so generic technology for working with them could well prove to be useful. Fortunately, every monoid induces an applicative functor, albeit in a slightly peculiar way:

```
newtype Accy o a = Acc{acc :: o}
```

Accy $o\ a$ is a *phantom type* (Leijen & Meijer, 1999)—its values have nothing to do with a, but it does yield the applicative functor of accumulating computations:

```
instance Monoid o ⇒ Applicative (Accy o) where
  pure _              = Acc ∅
  Acc o1 ⊛ Acc o2 = Acc (o1 ⊕ o2)
```

Now reduction or 'crushing' is just a special kind of traversal, in the same way as with any other applicative functor, just as Meertens suggested:

```
accumulate :: (Traversable t, Monoid o) ⇒ (a → o) → t a → o
accumulate f = acc · traverse (Acc · f)

reduce :: (Traversable t, Monoid o) ⇒ t o → o
reduce = accumulate id
```

Operations like flattening and concatenation become straightforward:

```
flatten :: Tree a → [a]          concat :: [[a]] → [a]
flatten = accumulate (:[])       concat = reduce
```

We can extract even more work from instance inference if we use the type system to distinguish different monoids available for a given datatype. Here, we use the disjunctive structure of Bool to test for the presence of an element satisfying a given predicate:

```
newtype Mighty = Might{might :: Bool}

instance Monoid Mighty where
  ∅                       = Might False
  Might x ⊕ Might y = Might (x ∨ y)

any :: Traversable t ⇒ (a → Bool) → t a → Bool
any p = might · accumulate (Might · p)
```

Now any · (≡) behaves just as the elem function for lists, but it can also tell whether a variable from v occurs free in an Exp v. Of course, Bool also has a conjunctive Musty structure, which is just as easy to exploit.

5 Applicative versus Monad?

We have seen that every Monad can be made Applicative via return and ap. Indeed, two of our three introductory examples of applicative functors involved the IO monad and the environment monad $(\rightarrow)\ env$. However the Applicative structure we defined on lists is not monadic, and nor is Accy o (unless o is the trivial one-point monoid): return can deliver $\emptyset$, but if you try to define

```
(>>=) :: Accy o a → (a → Accy o b) → Accy o b
```

you'll find it tricky to extract an a from the first argument to supply to the second—all you get is an o. The $\circledast$ for Accy o is not the ap of a monad.

So now we know: there are strictly more Applicative functors than Monads. Should we just throw the Monad class away and use Applicative instead? Of course not! The reason there are fewer monads is just that the Monad structure is more powerful. Intuitively, the $(\gg\!=) :: m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b$ of some Monad m allows the value returned by one computation to influence the choice of another, whereas $\circledast$ keeps

the structure of a computation fixed, just sequencing the effects. For example, one may write

```
miffy :: Monad m ⇒ m Bool → m a → m a → m a
miffy mb mt me = do
  b ← mb
  if b then mt else me
```

so that the value of *mb* will choose between the *computations* *mt* and *me*, performing only one, whilst

```
iffy :: Applicative f ⇒ f Bool → f a → f a → f a
iffy fb ft fe = ⟦cond fb ft fe⟧ where
  cond b t e = if b then t else e
```

performs the effects of all three computations, using the value of *fb* to choose only between the *values* of *ft* and *fe*. This can be a bad thing: for example,

$$\mathsf{iffy}\ [\![\mathsf{True}]\!]\ [\![t]\!]\ \mathsf{Nothing} = \mathsf{Nothing}$$

because the 'else' computation fails, even though its value is not needed, but

$$\mathsf{miffy}\ [\![\mathsf{True}]\!]\ [\![t]\!]\ \mathsf{Nothing} = [\![t]\!]$$

However, if you are working with miffy, it is probably because the condition is an expression with effectful components, so the idiom syntax provides quite a convenient extension to the monadic toolkit:

$$\mathsf{miffy}\ [\![(\leqslant)\ \mathit{getSpeed}\ \mathit{getSpeedLimit}]\!]\ \mathit{stepOnIt}\ \mathit{checkMirror}$$

The moral is this: if you've got an Applicative functor, that's good; if you've also got a Monad, that's even better! And the dual of the moral is this: if you want a Monad, that's good; if you only want an Applicative functor, that's even better!

One situation where the full power of monads is not always required is parsing, for which Röjemo (1995) proposed a interface including the equivalents of pure and '⊛' as an alternative to monadic parsers (Hutton & Meijer, 1998). Several ingenious non-monadic implementations have been developed by Swierstra and colleagues (Swierstra & Duponcheel, 1996; Baars *et al.*, 2004). Because the structure of these parsers is independent of the results of parsing, these implementations are able to analyse the grammar lazily and generate very efficient parsers.

Composing applicative functors The weakness of applicative functors makes them easier to construct from components. In particular, although only certain pairs of monads are composable (Barr & Wells, 1984), the Applicative class is *closed under composition*,

```
newtype (f ∘ g) a = Comp{comp :: (f (g a))}
```

just by lifting the inner Applicative operations to the outer layer:

```
instance (Applicative f, Applicative g) ⇒ Applicative (f ∘ g) where
  pure x                = Comp ⟦(pure x)⟧
  Comp fs ⊛ Comp xs = Comp ⟦(⊛) fs xs⟧
```

As a consequence, the composition of two monads may not be a monad, but it is certainly applicative. For example, both Maybe ∘ IO and IO ∘ Maybe are applicative: IO ∘ Maybe is an applicative functor in which computations have a notion of 'failure' and 'prioritised choice', even if their 'real world' side-effects cannot be undone. Note that IO and Maybe may also be composed as monads (though not vice versa), but the applicative functor determined by the composed monad differs from the composed applicative functor: the binding power of the monad allows the second IO action to be *aborted* if the first returns a failure.

We began this section by observing that Accy o is not a monad. However, given Monoid o, it can be defined as the composition of two applicative functors derived from monads—which two, we leave as an exercise.

Accumulating exceptions The following type may be used to model exceptions:

```
data Except err a = OK a | Failed err
```

A Monad instance for this type must abort the computation on the first error, as there is then no value to pass to the second argument of '⪢='. However with the Applicative interface we can continue in the face of errors:

```
instance Monoid err ⇒ Applicative (Except err) where
  pure                          = OK
  OK f ⊛ OK x                   = OK (f x)
  OK f ⊛ Failed err             = Failed err
  Failed err ⊛ OK x             = Failed err
  Failed err1 ⊛ Failed err2     = Failed (err1 ⊕ err2)
```

This could be used to collect errors by using the list monoid (as in unpublished work by Duncan Coutts), or to summarise them in some way.

6 Applicative functors and Arrows

To handle situations where monads were inapplicable, Hughes (2000) defined an interface that he called *arrows*, defined by the following class with nine axioms:

```
class Arrow (⇝) where
  arr   :: (a → b) → (a ⇝ b)
  (⋙)   :: (a ⇝ b) → (b ⇝ c) → (a ⇝ c)
  first :: (a ⇝ b) → ((a, c) ⇝ (b, c))
```

Examples include ordinary '→', Kleisli arrows of monads and comonads, and stream processors. Equivalent structures called *Freyd-categories* had been independently developed to structure denotational semantics (Power & Robinson, 1997).

There are similarities to the Applicative interface, with arr generalising pure. As with '⊛', the '⋙' operation does not allow the result of the first computation to affect the choice of the second. However it does arrange for that result to be fed to the second computation.

By fixing the first argument of an arrow type, we obtain an applicative functor, generalising the environment functor we saw earlier:

```
newtype EnvArrow (⇝) env a = Env (env ⇝ a)

instance Arrow (⇝) ⇒ Applicative (EnvArrow (⇝) env) where
  pure x           = Env (arr (const x))
  Env u ⊛ Env v = Env (u△v ⋙ arr (λ(f, x) → f x))
    where u△v = arr dup ⋙ first u ⋙ arr swap ⋙ first v ⋙ arr swap

dup a = (a, a)
swap (a, b) = (b, a)
```

In the other direction, each applicative functor defines an arrow constructor that adds static information to an existing arrow:

```
newtype StaticArrow f (⇝) a b = Static (f (a ⇝ b))

instance (Applicative f, Arrow (⇝)) ⇒ Arrow (StaticArrow f (⇝)) where
  arr f                 = Static ⟦(arr f)⟧
  Static f ⋙ Static g = Static ⟦(⋙) f g⟧
  first (Static f)      = Static ⟦first f⟧
```

To date, most applications of the extra generality provided by arrows over monads have been either various forms of process, in which components may consume multiple inputs, or computing static properties of components. Indeed one of Hughes's motivations was the parsers of Swierstra and Duponcheel (1996). It may turn out that applicative functors are more convenient for applications of the second class.

7 Applicative functors, categorically

The Applicative class features the asymmetrical operation '⊛', but there is an equivalent symmetrical definition.

```
class Functor f ⇒ Monoidal f where
  unit :: f ()
  (⋆)  :: f a → f b → f (a, b)
```

These operations are clearly definable for any Applicative functor:

```
unit    :: Applicative f ⇒ f ()
unit    = pure ()
(⋆)     :: Applicative f ⇒ f a → f b → f (a, b)
fa ⋆ fb = ⟦(,) fa fb⟧
```

Moreover, we can recover the Applicative interface from Monoidal as follows:

```
pure      :: Monoidal f ⇒ a → f a
pure x    = fmap (λ_ → x)        unit
(⊛)       :: Monoidal f ⇒ f (a → b) → f a → f b
mf ⊛ mx = fmap (λ(f, x) → f x) (mf ⋆ mx)
```

The laws of Applicative given in Section 2 are equivalent to the usual Functor laws, plus the following laws of Monoidal:

naturality of $\star$	$\mathsf{fmap}\ (f \times g)\ (u \star v)$	$=$	$\mathsf{fmap}\ f\ u \star \mathsf{fmap}\ g\ v$
left identity	$\mathsf{fmap}\ \mathsf{snd}\ (\mathsf{unit} \star v)$	$=$	v
right identity	$\mathsf{fmap}\ \mathsf{fst}\ (u \star \mathsf{unit})$	$=$	u
associativity	$\mathsf{fmap}\ \mathsf{assoc}\ (u \star (v \star w))$	$=$	$(u \star v) \star w$

for the functions

$$(\times) :: (a \to b) \to (c \to d) \to (a, c) \to (b, d)$$
$$(f \times g)\ (x, y) = (f\ x, g\ y)$$
$$\mathsf{assoc} :: (a, (b, c)) \to ((a, b), c)$$
$$\mathsf{assoc}\ (a, (b, c)) = ((a, b), c)$$

Fans of category theory will recognise the above laws as the properties of a *lax monoidal functor* for the monoidal structure given by products. However the functor composition and naturality equations are actually stronger than their categorical counterparts. This is because we are working in a higher-order language, in which function expressions may include variables from the environment, as in the above definition of pure for Monoidal functors. In the first-order language of category theory, such data flow must be explicitly plumbed using functors with *tensorial strength*, an arrow:

$$t_{AB} : A \times F\,B \longrightarrow F\,(A \times B)$$

satisfying standard equations. The natural transformation m corresponding to '$\star$' must also respect the strength:

$$\begin{array}{ccc}
(A \times B) \times (F\,C \times F\,D) & \cong & (A \times F\,C) \times (B \times F\,D) \\
{\scriptstyle (A \times B) \times m}\downarrow & & \downarrow{\scriptstyle t \times t} \\
(A \times B) \times F\,(C \times D) & & F\,(A \times C) \times F\,(B \times D) \\
{\scriptstyle t}\downarrow & & \downarrow{\scriptstyle m} \\
F\,((A \times B) \times (C \times D)) & \cong & F\,((A \times C) \times (B \times D))
\end{array}$$

Note that B and $F\,C$ swap places in the above diagram: strong naturality implies commutativity with pure computations.

Thus in categorical terms applicative functors are *strong lax monoidal functors*. Every strong monad determines two of them, as the definition is symmetrical. The Monoidal laws and the above definition of pure imply that pure computations commute past effects:

$$\mathsf{fmap}\ \mathsf{swap}\ (\mathsf{pure}\ x \star u) = u \star \mathsf{pure}\ x$$

The proof (an exercise) makes essential use of higher-order functions.

8 Conclusions

We have identified Applicative functors, an abstract notion of effectful computation lying between Arrow and Monad in strength. Every Monad is an Applicative functor, but significantly, the Applicative class is closed under composition, allowing computations such as accumulation in a Monoid to be characterised in this way.

Given the wide variety of Applicative functors, it becomes increasingly useful to abstract Traversable functors—container structures through which Applicative actions may be threaded. Combining these abstractions yields a small but highly generic toolkit whose power we have barely begun to explore. We use these tools by writing types that not merely structure the *storage* of data, but also the *properties* of data that we intend to exploit.

The explosion of categorical structure in functional programming: monads, comonads, arrows and now applicative functors should not, we suggest, be a cause for alarm. Why should we not profit from whatever structure we can sniff out, abstract and re-use? The challenge is to avoid a chaotic proliferation of peculiar and incompatible notations. If we want to rationalise the notational impact of all these structures, perhaps we should try to recycle the notation we already possess. Our $[\![f\ u_1\ \ldots\ u_n]\!]$ notation does minimal damage, showing when the existing syntax for applicative programming should be interpreted with an effectful twist.

Acknowledgements McBride is funded by EPSRC grant EP/C512022/1. We thank Thorsten Altenkirch, Duncan Coutts, Jeremy Gibbons, Peter Hancock, Simon Peyton Jones, Doaitse Swierstra and Phil Wadler for their help and encouragement.

References

Baars, A.I., Löh, A., & Swierstra, S.D. (2004). Parsing permutation phrases. *Journal of functional programming*, **14**(6), 635–646.

Barr, Michael, & Wells, Charles. (1984). *Toposes, triples and theories.* Grundlehren der Mathematischen Wissenschaften, no. 278. New York: Springer. Chap. 9.

Fridlender, Daniel, & Indrika, Mia. (2000). Do we need dependent types? *Journal of Functional Programming*, **10**(4), 409–415.

Hughes, John. (2000). Generalising monads to arrows. *Science of computer programming*, **37**(May), 67–111.

Hutton, Graham, & Meijer, Erik. (1998). Monadic parsing in Haskell. *Journal of functional programming*, **8**(4), 437–444.

Leijen, Daan, & Meijer, Erik. 1999 (Oct.). Domain specific embedded compilers. *2nd conference on domain-specific languages (DSL).* USENIX, Austin TX, USA. Available from `http://www.cs.uu.nl/people/daan/papers/dsec.ps`.

Meertens, Lambert. 1998 (June). Functor pulling. *Workshop on generic programming (WGP'98).*

Peyton Jones, Simon (ed). (2003). *Haskell 98 language and libraries: The revised report.* Cambridge University Press.

Power, John, & Robinson, Edmund. (1997). Premonoidal categories and notions of computation. *Mathematical structures in computer science*, **7**(5), 453–468.

Röjemo, Niklas. (1995). *Garbage collection and memory efficiency.* Ph.D. thesis, Chalmers.

Swierstra, S. Doaitse, & Duponcheel, Luc. (1996). Deterministic, error-correcting combinator parsers. *Pages 184–207 of:* Launchbury, John, Meijer, Erik, & Sheard, Tim (eds), *Advanced functional programming.* LNCS, vol. 1129. Springer.

Wadler, Philip. (1985). How to replace failure by a list of successes. *Pages 113–128 of:* Jouannaud, Jean-Pierre (ed), *Functional programming languages and computer architecture.* LNCS, vol. 201. Springer.